T.P. 6 : Gestion de la mémoire (Partie 1)

On désire réaliser un gestionnaire de mémoire qu'on implémentera sur trois niveaux :

1. Gestion physique de la mémoire. Cette mémoire sera représentée par un segment de mémoire partagée.
 2. L'interface entre la mémoire et les processus. Cette interface correspond en fait au noyau et aux appels-système.
 3. Les processus clients.
1. Cette partie aura pour rôle d'allouer un ensemble de blocs contigus, de libérer ces blocs ou d'allouer un ou plusieurs blocs supplémentaires. On propose d'écrire les trois fonctions suivantes :
 - `alloue(taille)`. alloue un ensemble de blocs pour un processus. Cette fonction retourne ensuite un identificateur sur la liste des blocs alloués.
 - `libere(id)`. Cette fonction libère les blocs à partir de `id` liés à un processus.
 - `bloc_supp(id,nb_supp)`. Cette fonction alloue `nb_supp` blocs supplémentaires. S'il existe de l'espace libre à l'adresse `id`, alors ces blocs sont alloués immédiatement après et la fonction retourne `id`, sinon le gestionnaire recherche un nombre blocs libres suffisant et recopie les anciens blocs à la nouvelle adresse avant de retourner le nouvel identificateur.
 - `get_nb_bloc(id)`. Retourne le nombre de blocs alloués pour un processus à l'adresse `id`

On mémorise par liste chaînée les informations suivantes :

- `etat` : libre ou occupée.
- numéro de bloc
- `taille`
- `pointeur suivant`

Deux variantes sont à envisager :

- (a) Une seule liste gerant les zones libres et occupées.
- (b) Deux listes distinctes.

On aura intérêt à chaîner de manière double ces deux listes. Nous proposons différents algorithmes basés sur une gestion par liste chaînée :

- La méthode de la première zone libre.
- La méthode du meilleur ajustement.
- La méthode du plus grand résidu.

2. Cette partie correspond au noyau. Elle permet d'allouer le segment de code et de données (demande auprès du gestionnaire de mémoire `mmem`). et de gérer l'allocation dynamique à l'intérieur de ce segment de données pour chacun des processus. On retrouvera les fonctions disponible dans `mmem` ainsi que les fonctions de gestion de l'allocation dynamique (allocation reallocation, liberation). Dans un premier temps, on n'implantera que la fonction `malloc()`. On utilisera la structure suivante :
 - `pid`,
 - `numero de bloc (id)`,
 - `taille du code (en octet)`,

– taille des données (en octet)

De plus, on implémentera les deux appels-système (provenant des processus) `put_mem(adresse,valeur,taille)` et `get_mem(adresse,taille)`. L'adresse est pour le processus le résultat du `malloc`. Pour ces deux fonctions, on vérifiera que la zone de données a bien été allouée et ne sort pas de l'espace d'adressage du processus.

On pourra utiliser comme structure de données pour gérer l'allocation dynamique, une liste de liste (une liste de processus et pour chacun d'eux, une liste des allocations dynamiques).

3. Les processus devront utiliser les appels système suivant :

– `debut()`. Allocation de la mémoire pour le processus. (On pourra utiliser le code source comme code du programme).

– `fin()`. Desallocation du processus.

– `mmalloc(taille)`. La taille à allouer est en octet. Cette fonction retourne l'adresse de la partie allouée. C'est un offset par rapport au début de l'espace d'adressage.

– `put_mem(adr,val,lg)` positionne à l'adresse précisée par `adr` la valeur `val` sur la taille `lg`. L'adresse est ici encore un offset.

– `get_mem(adr,val,lg)` récupère à l'adresse précisée par `adr` la valeur que l'on place dans `val` sur la taille `lg`.

– On pourra implémenter de manière optionnel les fonctions `free(id)` et `realloc(id,taille)`.

La communication entre ce module et celui de gestion des processus se fait par des files de messages.